# Superoptimization of Memory Subsystems

**Joseph G. Wingbermuehle**
**Ron K. Cytron**
**Roger D. Chamberlain**

Joseph G. Wingbermuehle, Ron K. Cytron, and Roger D. Chamberlain, "Superoptimization of Memory Subsystems," in *Proc. of ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES), June 2014, pp 145-154.

Dept. of Computer Science and Engineering
Washington University in St. Louis

# Superoptimization of Memory Subsystems [*]

Joseph G. Wingbermuehle Ron K. Cytron Roger D. Chamberlain
Washington University in St. Louis
{wingbej,cytron,roger}@wustl.edu

## Abstract

The disparity in performance between processors and main memories has led computer architects to incorporate large cache hierarchies in modern computers. Because these cache hierarchies are designed to be general-purpose, they may not provide the best possible performance for a given application. In this paper, we determine a memory subsystem well suited for a given application and main memory by discovering a memory subsystem comprised of caches, scratchpads, and other components that are combined to provide better performance. We draw motivation from the *superoptimization* of instruction sequences, which successfully finds unusually clever instruction sequences for programs. Targeting both ASIC and FPGA devices, we show that it is possible to discover unusual memory subsystems that provide performance improvements over a typical memory subsystem.

***Categories and Subject Descriptors*** C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; B.1.4 [*CONTROL STRUCTURES AND MICRO-PROGRAMMING*]: Optimization

***Keywords*** Superoptimization; Cache

## 1. Introduction

Memory accesses are the primary bottleneck for many applications [34]. In an effort to alleviate this bottleneck, modern computers typically use large cache hierarchies between compute resources, such as general-purpose processors or field-programmable gate arrays (FPGAs), and main memory. These cache hierarchies form a memory subsystem that connects the compute resources to the main memory with the goal of improving the performance of an application by exploiting common properties of memory accesses, such as temporal and spatial localities.

Most memory subsystems built today are designed to be general-purpose by providing good overall memory performance for a wide variety of applications. This is exemplified by the fact that most computers today have a fixed cache hierarchy. However, because of this generality, such memory subsystems do not necessarily provide the best possible performance for a particular application and main memory. Further, a general-purpose cache may use more on-chip resources than necessary for a particular application. With this in mind, we set out to investigate whether it is possible to design application-specific memory subsystems that provide better performance than a general-purpose memory subsystem. Such memory subsystems could take advantage of specific properties of both the application and main memory to provide better performance than might be possible with a general-purpose memory subsystem.

In addition to the improved performance, if it were possible to discover a good memory subsystem automatically, the manual and laborious process of optimizing an application for a particular memory subsystem may not be necessary. Although both cache-aware [28] and cache-oblivious [9] algorithms and data structures exist, it remains a relatively difficult task to make an arbitrary application perform well for a given memory subsystem and main memory.

Rather than focus exclusively on selecting cache parameters for a fixed cache hierarchy, we widen the search space to include components other than caches, such as scratchpad memories and address transformations, which are not typically found in memory subsystems. In addition, we consider total access time rather than relying solely on cache misses. This allows us to customize the memory subsystem to take advantage of certain properties of the main memory, such as burst behavior.

This work draws motivation from superoptimization [20], which has been used successfully in GCC to find short instruction sequences [13]. More recently, the concept of superoptimization has been extended using stochastic search to explore larger code segments [27]. The concept of *superoptimization* in those works is to try many instruction sequences with the hope of discovering a new sequence that is shorter or faster than other, functionally equivalent, instruction sequences. However, rather than superoptimizing instruction sequences, we are interested in superoptimizing memory subsystems.

Unfortunately, modern computer systems do not have configurable memory subsystems. Indeed, even in the embedded space, there are often limitations on how much a memory subsystem can be modified. However, it is conceivable that general-purpose computers might introduce more memory subsystem flexibility in the future, especially if such flexibility were able to provide a significant performance advantage. Moreover, flexible memory subsystems exist today for applications deployed on FPGAs and application-specific integrated circuits (ASICs).

In this work, we target both an FPGA and an ASIC process. The FPGA we target is a Xilinx Virtex-7 running at 250 MHz. An FPGA is a type of reconfigurable hardware consisting of configurable logic gates (formed from look-up tables), registers, and a routing matrix. In addition, modern FPGAs typically have additional resources, such as block RAMs and hardware multipliers. Block RAMs are configurable memories, which make it possible to implement diverse memories efficiently on an FPGA device. These

[*] This work is supported by the National Science Foundation under grants CNS-09095368 and CNS-0931693.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*LCTES '14*, June 12–13, 2014, Edinburgh, UK.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2877-7/14/06…$15.00.
http://dx.doi.org/10.1145/2597809.2597816

block RAMs are of particular interest to us since we can use them in our memory subsystems.

Block RAMs are typically some fixed size in terms of storage bits, but with a configurable aspect ratio. For example, on our target device, block RAMs are 72 bits wide and 512 entries deep. Such a block RAM can be used to implement other aspect ratios, such as 36 bits by 1024 entries, 18 bits by 2048 entries, etc. By using multiple block RAMs, one can create larger memories as well.

To demonstrate the generality of our approach, we target a 45 nm ASIC process in addition to the FPGA target. In both the FPGA and ASIC cases, we assume a DDR3 main memory, however, we note that designing custom main memory models for our superoptimizer is straightforward.

To superoptimize a memory subsystem for an application, we use a memory address trace from a representative run of the application. We then generate candidate memory subsystems from a neighborhood of memory subsystems around the previous proposal and simulate the address trace. Finally, we either accept or reject the proposed memory subsystem based on its performance. This process repeats until we give up searching.

Here we build upon earlier work on application-specific memory subsystems [33] by evaluating a wider variety of benchmark applications, using a different approach to meta-heuristic optimization, performing an explicit model validation, and including support for an ASIC target. In addition, we use a more realistic model for the main memory. Finally, we show that it is possible to exploit information available in the memory traces to discover better memory subsystems faster.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes how we perform the optimization and as well as the experimental setup. Section 4 shows our experimental results. Section 5 provides a discussion of the results and what they mean. Finally, Section 6 concludes.

## 2. Related Work

Although there is much related work on design space exploration [16, 19], the ability to change completely the memory subsystem for a specific application and main memory subsystem distinguishes this work from most previous work.

Techniques for improving the performance of memory subsystems at the software level include a profiling approach used to guide the placement of variables in the virtual address space to decrease cache conflicts and improve locality [3]. Compiler optimizations have been used to improve data locality across loop iterations [4]. Reorganization and cache-conscious memory allocation have been explored [7]. Finally, splitting and reordering of structures have been explored in the interest of improving cache behavior [6]. These approaches do not consider altering the memory subsystem at the hardware level.

At a higher level, there are approaches to application design that focus on improving cache performance. In particular, cache-aware algorithms [28] attempt to take advantage of a particular cache structure. Likewise, the performance of cache-oblivious algorithms [9] is asymptotically optimal on an ideal cache hierarchy. Unfortunately, ideal cache hierarchies do not exist and, further, these techniques are specific to a very small set of algorithms.

Much related work exists with respect to tuning the parameters of a fixed memory subsystem dynamically. Methods to change the size and associativity of a cache hierarchy dynamically have been explored [2, 29] and the ability to disable various levels of a multi-level cache in the interest of reducing latency and reducing power consumption has been considered [5]. Finally, adjusting the size of cache lines dynamically to lower the cache miss rate has been considered [31]. Unlike our method, these approaches are dynamic, but limited in the amount of variation supported by the memory subsystem. Techniques such as these are complementary to our work as they could be added to the set of memory subsystem components that our optimizer has at its disposal.

There are also many approaches to tuning cache parameters statically. A method for selecting cache parameters analytically has been described for a single-level cache [10]. In addition, heuristic methods for selecting the parameters of a two-level cache have been presented [11, 12]. Unlike our approach, these approaches consider a much smaller search space. However, we note that it may be possible to use such techniques to speed up the superoptimization process.

Non-traditional memory subsystems have been proposed. For example, a victim cache [17] cache is a small, fully-associative cache structure used to store recently evicted items from a larger cache with low associativity. The use of such caches in embedded applications has been explored [35]. The combination of a scratchpad and cache has been considered [23, 25]. Further, the combination of multiple caching techniques including split caches has been considered [21]. Each of these works presents a particular memory subsystem. Our work, on the other hand, attempts to discover memory subsystems with arbitrary structure. Therefore, it is possible that our superoptimizer would discover similar structures if provided the necessary memory subsystem components.

Finally, there is complementary work in the area of hardware synthesis. For example, LEAP scratchpads [1] provide multiple logical memories to an FPGA backed by a single main memory. In such a system, our work could be used to discover the best use of the on-chip memory resources to improve performance. Likewise, our system could be used to find more suitable memory subsystems in high-level synthesis tools such as CHiMPS [24] and ScalaPipe [32].

## 3. Method

Our goal is to discover a memory subsystem that offers the best possible performance for a given application and main memory. To do this, we propose and evaluate candidate memory subsystems. We can then view the discovery of good memory subsystems as a classical optimization problem with an objective function and constraints. Here, we seek to minimize the total memory access time constraining the on-chip memory used. It would be possible to use other objective functions, however, such as minimization of energy use or minimization of writes to main memory.

We consider both an FPGA device and an ASIC process for deployment. When targeting an FPGA device, we constrain the optimization by specifying the maximum number of block RAMs (BRAMs) that can be used in the memory subsystem as well as the minimum operating frequency. During the optimization process, we synthesize each component individually and store the results so that they can be reused. The optimization proceeds assuming that, when combining memory components, the maximum clock frequency does not change and that the number of block RAMs is added together. Although the assumption about the number of required block RAMs is conservative, the assumption about the frequency remaining the same is not conservative. Therefore, once the optimization is complete, we validate that the discovered memory subsystem will run at the target frequency and fit on the target device by synthesizing the complete memory subsystem.

When targeting an ASIC process, we constrain the optimization by specifying the maximum chip area dedicated to the memory subsystem and assume that the system runs at a fixed clock frequency. We use the CACTI [30] program to determine the access time, cycle time, and area required for each memory component. Full synthesis, however, is not performed for the ASIC target.

| Component | Description | Parameters ($n \in \mathbb{Z}_+$) | Latency (cycles) |
|---|---|---|---|
| Cache | Parameterizable cache | Line size ($word\_size \times 2^n$)<br>Line count ($2^n$)<br>Associativity ($1 \ldots line\_count$)<br>Replacement (LRU, MRU, FIFO, PLRU)<br>Write policy (write-back, write-through) | 3 |
| Offset | Address offset | Value ($\pm n$) | 0 |
| Prefetch | Stride prefetcher | Stride ($\pm n \times word\_size$) | 0 |
| Rotate | Rotate address transform | Value ($\pm n$) | 0 |
| Scratchpad | Scratchpad memory | Size ($word\_size \times 2^n$) | 2 |
| Split | Split memory | Location ($n \times word\_size$) | 0 |
| XOR | XOR address transform | Value ($n$) | 0 |

**Table 1.** Memory Subsystem Components

### 3.1 Address Traces

We use address traces to evaluate the performance of a particular memory subsystem for an application. We consider two types of traces: traces gathered from real applications and traces gathered from custom kernels. To gather the address traces for the application benchmarks, we use a modified version of the Valgrind [22] *lackey* tool. This allows us to obtain concise address traces for applications that contain only data accesses (reads, writes, and modifies). We ignore instruction accesses since the instructions would likely be stored in a separate memory.

For both the application benchmarks and kernel benchmarks, we ignore the notion of processing time in the trace; our focus is exclusively on memory performance. Because there is no notion of processing time, however, certain memory subsystem components, such as prefetchers, are unlikely to be useful. Introducing processing time is possible, but to do so would require a specific implementation of the application, which would make the results less general.

All of the address traces contain virtual (instead of physical) addresses and are gathered for 32-bit versions of the benchmark applications. To evaluate a general-purpose memory subsystem, the physical addresses are important since some levels of cache use physical addresses to avoid problems when context switching. However, we note that our memory subsystem is specific to the application and, therefore, using virtual addresses is appropriate. We leave the problem of sharing the same memory resources among multiple applications or kernels as future work.

### 3.2 Simulation

Given an address trace for an application and a candidate memory subsystem, we use a custom memory system simulator to determine how many cycles are spent performing memory accesses. Our simulator is capable of simulating the memory subsystem components shown in Table 1, which also shows the latency in cycles required for the FPGA implementation.

For caches, the simulator supports four replacement policies. The supported policies include least-recently used (LRU), most-recently used (MRU), first-in first-out (FIFO), and pseudo-least-recently used (PLRU). The PLRU policy approximates the LRU policy by using a single *age* bit per cache way rather than $\lg n$ age bits, where $n$ is the associativity of the cache. With the PLRU policy, the first way where the age bit is not set is selected for replacement. Upon access, the age bit for the accessed way is set and when all age bits are set for a set, all but the accessed age bit are cleared.

The offset, rotate, and xor components in Table 1 are address transformations. The *offset* component adds the specified value to the address. The *rotate* component rotates the bits of the address that select the word left by the specified amount (the bits that select the byte within the word remain unchanged). Note that for a 32-bit address with a 4-byte word, $32 - \lg 4 = 30$ bits are used to select the word. Finally, the *xor* component inverts the selected bits of the address.

| Parameter | Description | Value |
|---|---|---|
| Frequency | The I/O frequency of the DRAM | 400 MHz |
| CAS | Cycles select a column | 5 |
| RCD | Cycles from opening to read/write | 5 |
| RP | Cycles required to precharge a row | 5 |
| Page size | Size of a page in bytes | 1024 |
| Page count | Number of pages per bank | 65536 |
| Width | Channel width in bytes | 8 |
| Burst size | Number of columns per access | 4 |
| Page mode | Open or closed page mode | open |
| DDR | Double data rate | true |

**Table 2.** Main Memory Parameters

Other supported components include prefetch and split. The *prefetch* component performs an additional memory access after every memory access to the prefetch. This additional access reads the word with the specified distance from the original word that was accessed. Finally, the *split* component divides memory accesses between two memory subsystems based on address.

The communication between each of the memory components as well as the communication between the application and main memory is performed with 4-byte words. The bytes within the word are selected using a 4-bit mask to allow byte-addressing. The address bus is 30 bits, providing a 32-bit address space.

As presented here, the optimizer supports seven distinct subsystem components. However, adding additional components is simply a matter of adding a synthesizable HDL model of the component and a simulation model for the optimizer. Likewise, additional parameters can be added to the existing components. Unfortunately, adding additional components can make the optimization process take longer since more steps will be required.

In addition to simulating memory subsystems, our memory simulator is capable of simulating main memories with various properties, shown in Table 2. As is the case with the memory subsystems, it is possible to model main memories with other properties if required. For our purposes, we consider a DDR3-800D memory, whose properties are shown in Table 2.

### 3.3 Optimization

To guide the optimization process, we use a variant of *threshold acceptance* [8] called *old bachelor acceptance* [15]. Old bachelor acceptance is a Markov-chain Monte-Carlo (MCMC) stochastic hill-climbing technique similar to simulated annealing [18].

Using stochastic hill-climbing, one typically selects an initial state, $s_t = s_0$, and then generates a *proposal* state, $s^*$, in the neighborhood of the current state. The state is then either accepted, becoming $s_{t+1}$, or rejected. With threshold acceptance, the difference in cost between the current state, $s_t$, and the proposal state, $s^*$, is compared to a threshold, $T_t$, to determine if the proposal state should be accepted. Thus, we get the following expression for determining the next state:

$$s_{t+1} = \begin{cases} s^* & \text{if } c(s^*) < c(s_t) + T_t \\ s_t & \text{otherwise} \end{cases}$$

For our purposes, the state is a candidate memory subsystem and the cost function, $c(\cdot)$, is the total access time in cycles that the application will experience from memory accesses.

With threshold acceptance, the threshold is initialized to some relatively high value, $T_t = T_0$. The threshold is then lowered according a cooling schedule. The recommended schedule in [8] is $T_{t+1} = T_t - \Delta T_t$ where $\Delta \in (0, 1)$. Old bachelor acceptance generalizes this, allowing the threshold to be lowered when a state is accepted and raised when a state is not accepted. This allows the algorithm to escape areas of local optimality more easily. For our experiments, we used the following schedule:

$$T_{t+1} = \begin{cases} T_t - \Delta T_t & \text{if } c(s^*) < c(s_t) + T_t \\ T_t + \Delta T_t & \text{otherwise} \end{cases}$$

Because the evaluation of a state involves simulating a memory subsystem for an address trace, each state evaluation can take several minutes or even longer depending on the size of the trace. Further, to discover a good memory subsystem, the total number of states visited can be large, which can make the optimization process take a prohibitively long time.

To reduce the time required for superoptimization, we employ two techniques to speed up the process. First, we memoize the results of each state evaluation so that when revisiting a state we do not need to simulate the memory trace again. The second improvement is that we allow multiple superoptimization processes to run simultaneously sharing results, thereby allowing us to exploit multiple processor cores.

### 3.4 Neighborhood Generation

Our memory subsystem optimizer is capable of proposing candidate memory subsystems comprised of the structures shown in Table 1. These components can be combined in arbitrary ways leading to a huge search space limited only by the constraints. Recall that for the FPGA target, the constraints include the minimum clock frequency and the maximum number of BRAMs for the memory subsystem. For the ASIC target, the constraint is the area as reported from the CACTI tool.

Given a state, $s_t$, we compute a proposal state $s^*$ by performing one of the following actions:

1. Insert a new memory component to a random position,
2. Remove a memory component from a random position, or
3. Change a parameter of the memory component at a random position.

With MCMC algorithms such as simulated annealing and threshold acceptance, it is necessary that the generated proposal states be *ergodic*. Ergodicity means that it is possible to reach every state from any given state in a finite number of steps. It is easy to see that the above process is ergodic as actions 1 and 2 are capable of canceling each other and action 3 can cancel itself.

To ensure that any discovered memory subsystem is valid, we reject any memory subsystem that exceeds the constraints. However, there are other ways a memory subsystem may be invalid. First, because we support splitting between memory components by address, any address transformation occurring in a split must be inverted before leaving the split. To handle this, we always insert (or remove) both the transform and its inverse when inserting (or removing) an address transformation.

Another situation that can lead to an invalid memory subsystem is when a complex memory subsystem prevents the subsystem from achieving the required clock frequency on the FPGA device. Note that for an ASIC device we increase the number of cycles required to access the memory component. Although we synthesize each component for the FPGA target separately to prevent this, it is still possible that a combination of components prevent the complete memory subsystem from achieving the required clock frequency.

To prevent the optimizer from generating a memory subsystem that is unable to run at the required clock frequency, the optimizer keeps a rough estimate on the longest combinational path and prevents the path from becoming too long. Nevertheless, it is still possible that a particular superoptimized memory subsystem may not achieve the required clock frequency. Therefore, for the FPGA results, we synthesize the superoptimized memory subsystems to validate them.

### 3.5 Offset Selection Heuristic

Because the search space is so large, arbitrarily selecting addresses to segment the address space in a split component can be problematic. Therefore, rather than proposing arbitrary addresses for split offsets, we restrict the set of addresses to values that actually exist in the address trace. We do this by recording the address ranges that are used during the first evaluation of the trace for the initial state. To further improve these results, the addresses we generate are weighted such that those addresses at the ends of address ranges are more likely to be selected.

Given an address range of length $n$ that starts at $a$, addresses used for splits are selected according to the following algorithm:

$$A(a, n) = \begin{cases} a & \text{w.p. } 1/8 \\ a + n - 1 & \text{w.p. } 1/8 \\ A(a, \lceil n/2 \rceil) & \text{w.p. } 3/8 \\ A(a + \lfloor n/2 \rfloor, \lceil n/2 \rceil) & \text{w.p. } 3/8 \end{cases}$$

Here *w.p.* stands for “with probability”. Thus, there is a 12.5% chance of selecting the first address in the range, a 12.5% chance of selecting the last address, and a 75% chance of selecting an address between these two extremes.

### 3.6 Model Validation

To validate the simulation model used during the optimization process, our optimizer generates synthesizable VHDL that has the characteristics shown in Table 1. By synthesizing the VHDL, we can ensure that the discovered memory subsystem is able to run at the required frequency and fit on our target device. The synthesis targets a Xilinx Virtex-7 running at 250 MHz.

### 3.7 Benchmarks

We use a collection of six benchmarks from the MiBench benchmark suite [14] as well as four kernels for evaluation purposes. The MiBench benchmark suite contains single-threaded benchmarks for the embedded space that target a variety of application areas. For some benchmarks, the MiBench suite contains large and small versions. We chose the large version in the interest of obtaining larger memory traces.

The locally developed kernels include a kernel that performs random lookups in a hash table (hash), a kernel that performs

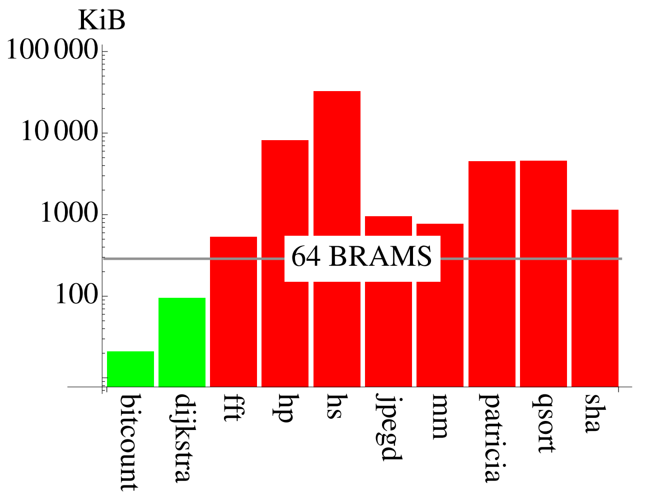


**Figure 1.** Working-Set Sizes

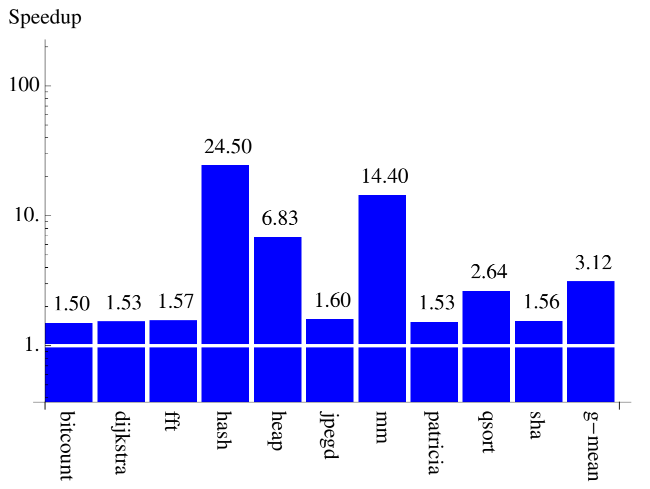


**Figure 2.** Best-case FPGA Speedup (Log Scale)

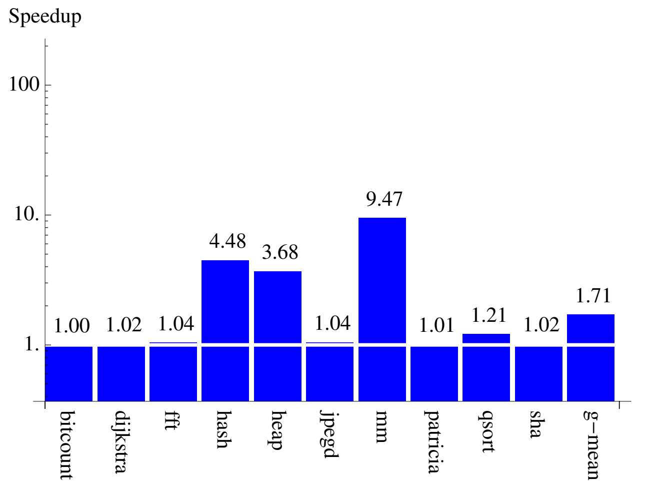


**Figure 3.** Realized FPGA Speedup (Log Scale)

matrix-matrix multiply (mm), a kernel that inserts and then removes items from a binary heap (heap), and a kernel that sorts an array of integers using the Quicksort algorithm (qsort). Rather than implement an application to perform these operations and use Valgrind to capture the address trace, the addresses traces for these kernels are generated directly during a simulation run, which allows us to avoid processing large trace files for the kernels.

Because we are optimizing the memory subsystem, the amount of memory accessed by each benchmark is important. If a particular benchmark accesses less memory than is available to the on-chip memory subsystem, then it should be possible to have all memory accesses occur in on-chip memory, though such a design may require clever address transformations. A graph of the total working-set size for each benchmark is shown in Figure 1.

In Figure 1, we see that there are two benchmarks, bitcount and dijkstra, that are small enough that all memory accesses could be mapped into 64 BRAMs, which is 2,359,296 bits, or 294,912 bytes. All of the other benchmarks are too large to fit completely within 64 BRAMs, which is the constraint on BRAMs we consider for the FPGA target.

For the 45nm ASIC process with an area constraint of 1mm$^2$, we can store a total of 379,392 bytes in a scratchpad according to our CACTI model. This means that, as with the FPGA, both the bitcount and dijkstra benchmarks are small enough to be mapped into a scratchpad, but all of the remaining benchmarks require too much memory.

## 4. Results

To evaluate the performance of our superoptimized memory subsystems, we compare the performance of the superoptimized memory subsystems against a baseline cache. For our baseline cache, we selected a cache that closely resembles the data cache in a Raspberry Pi [26]. This is a 64 KiB, 4-way set-associative write-back cache with 32-byte lines and a PLRU replacement policy. The FPGA implementation of this cache uses 16 BRAMs and meets our 250 MHz target frequency. According to CACTI, the 45nm ASIC implementation is 0.18mm$^2$ with a 1-cycle access time and a 3-cycle cycle time.

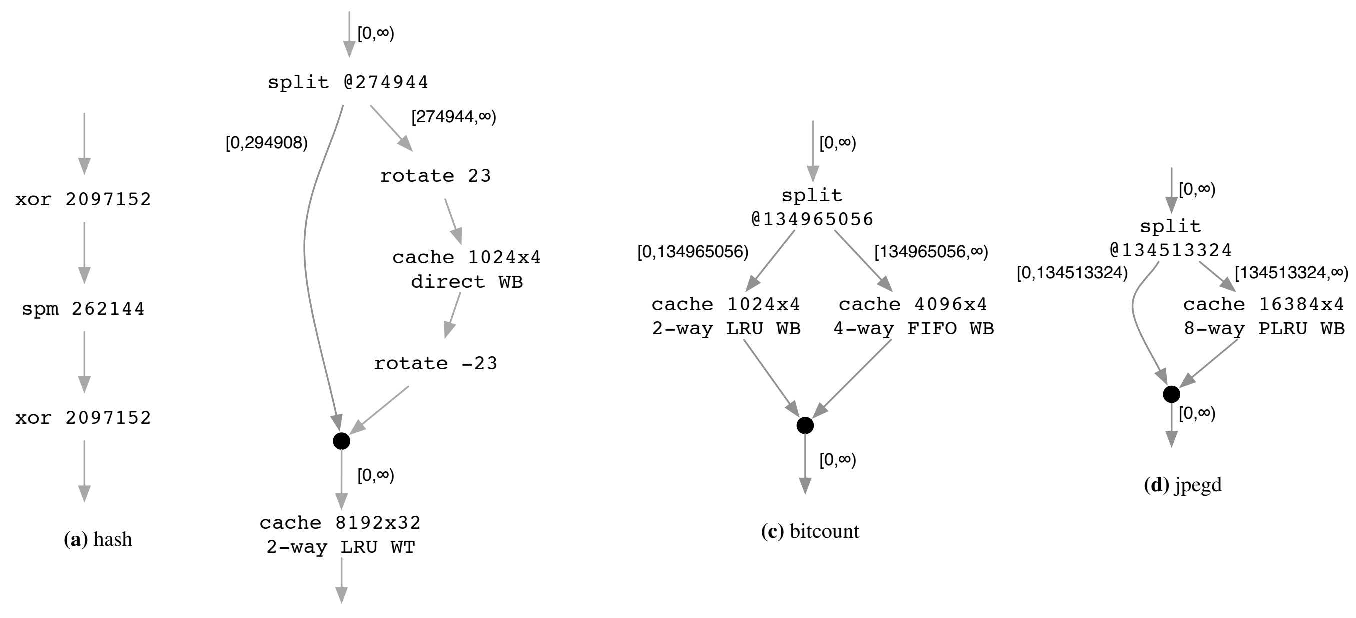
### 4.1 FPGA Results

For the first set of experiments, we target a Xilinx Virtex-7 with a target frequency of 250 MHz and a constraint of 64 BRAMs maximum. The main memory is assumed to be the DDR3 device whose properties are shown in Table 2.

The first question we attempt to answer is: how much better might we make the memory subsystem than the baseline cache? To determine this, we compare the performance of each benchmark to a "best-case" access time. For the best-case access time, we assume that all memory accesses hit in the fastest memory component available for each of our targets. For the FPGA target, this means that all accesses hit in a scratchpad and, therefore, take two cycles to complete. This best-case speedup for our benchmarks running on the FPGA target is shown in Figure 2.

The *g-mean* bar in Figure 2 represents the geometric mean. Assuming that we could somehow arrange for all of the memory accesses to hit in the scratchpad we would get a $3.12\times$ speedup over the baseline cache for the FPGA target. Note that, in reality, such a speedup is not possible since we do not have enough resources available to make all of the accesses hit in a scratchpad.

Figure 3 shows the speedup that the superoptimized memory subsystem provides over the baseline memory subsystem. Across the set of benchmark applications, the performance gain varies



**(a)** hash **(b)** mm **(c)** bitcount **(d)** jpegd

**Figure 4.** Superoptimized Memory Subsystems for the FPGA Target

from very little to over $9\times$ with a geometric mean speedup of $1.71\times$.

Although the results are not much better than the baseline memory subsystem, we note that for all of the benchmarks there was some improvement, though less than $1\%$ in a few cases. There are a few benchmarks, however, that exhibit substantial performance gain. The matrix-matrix multiply shows the best speedup of over $9\times$. Because the main memory is not much slower than the cache structures running on a 250 MHz FPGA fabric, we do not anticipate substantial gains for all of the applications (see Figure 2). A number of the discovered memory subsystems are, however, worth considering in more detail.

The first interesting memory subsystem we consider is the superoptimized memory subsystem for the hash benchmark, shown in Figure 4a. The hash benchmark performs random probes into a hash table containing 8,388,608 entries, each 4-bytes. This type of access pattern causes problems for caches due to the lack of locality. In Figure 4a, memory accesses enter the top and accesses to main memory come out the bottom. There are two address transformations and a 262,144-byte scratchpad. The first address transformation toggles a bit of the address. The transformed address then enters the scratchpad. The second transformation reverses the first transformation so that the addresses remain unchanged as they enter the main memory (recall that address transformations are always inserted and removed in pairs).

The reason that the address transformation is beneficial for the hash benchmark is due to the random accesses to the hash table being slightly unbalanced. Removing the address transformation results in a very slight decrease in performance. If we remove the scratchpad completely, there is again only a slight decrease in performance. Here we note that the speedup is primarily due to the removal of the cache, which serves only to cause overhead when there is no locality. The scratchpad speeds up some of the accesses, but only a small fraction.

Another interesting memory subsystem, which also provides the greatest performance improvement, is discovered for mm: the matrix-matrix multiply benchmark. This benchmark performs a matrix-matrix multiply using the naive $O(n^3)$ algorithm with 256-by-256 matrices. Each element of the matrix is 4 bytes. The superoptimized memory subsystem for this benchmark is shown in Figure 4b. In the superoptimized memory subsystem for the mm benchmark, memory accesses enter the top and are then split, with accesses below address 274944 going directly to a 262,144-byte cache at the bottom of Figure 4b and accesses to addresses above and including 274944 going to a separate memory subsystem before going to the 262,144-byte cache. For accesses to addresses above and including 274944, first the bits of the address that select the word are rotated left by 23 bits. The accesses then enter a 4,096-byte, direct-mapped cache, and finally, the address is rotated right by 23 bits before entering the larger cache.

To understand why the memory subsystem for the mm benchmark provides such good performance, we consider the way the memory is organized for the benchmark. There are 3 matrices: two sources and a destination. The first source matrix, which is accessed in row-major order, is stored in addresses 0 through 262140. The second source matrix, which is accessed in column-major order, is stored at addresses 262144 through 524284. Finally, the destination matrix is stored at addresses 524288 through 786428.

With this memory organization in mind, we note that the address split moves most accesses for the second source matrix as well as the destination matrix into a separate memory subsystem. Within this subsystem, the addresses are transformed and then routed to a cache. Given that the second source matrix is accessed in column-major order, for the first column, we access $00040000_{16}, 00040400_{16}, \ldots 0007FC00_{16}$ for the first column, then $00040004_{16}, 00040404_{16}, \ldots 0007FC04_{16}$ for the second column, and so on. However, after the split and address transformation, the addresses from the perspective of the 1024 entry cache look about like this: $00000000_{16}, 00000008_{16}, \ldots 00000FF8_{16}$ for the first column, $01000000_{16}, 01000008_{16}, \ldots 01000FF8_{16}$ for the second column, and so on. The result is that each column of the matrix is cached and can be reused 256 times before the next column is required.

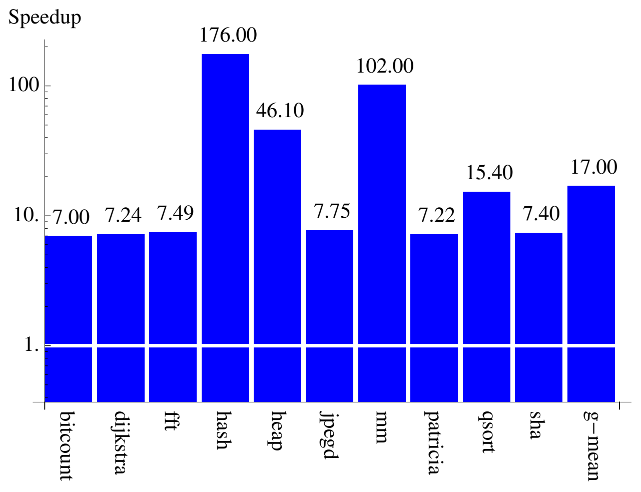


**Figure 6.** Best-case ASIC Speedup (Log Scale)

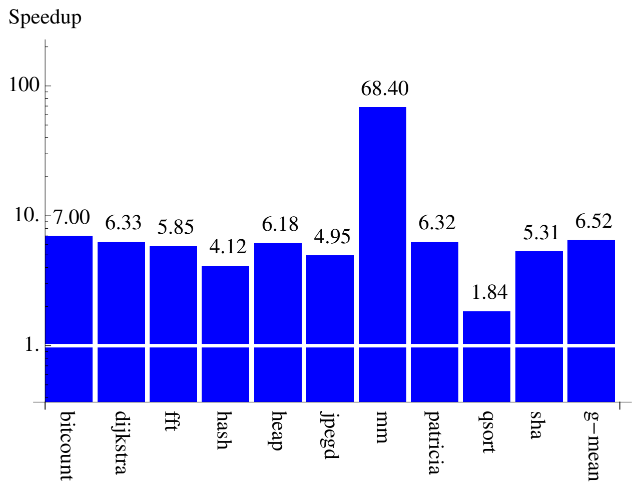


**Figure 7.** Realized ASIC Speedup (Log Scale)

Note that due to the layout of the matrices, one would expect that the ideal address for the split would be 262144 instead of 274944. Indeed, changing the split address results in a 0.46% improvement in performance. Thus, running the superoptimizer longer would likely result in an even better memory subsystem. Further, this implies that there may be better ways to propose offsets for splits.

A final observation about the memory subsystem for the mm benchmark is the large cache after the split. This cache has 32-byte cache lines, which allows it to prefetch values for the source matrix. Also, the cache is write-through rather than write-back, which prevents cache pollution due to writes to the destination matrix.

The memory subsystem discovered for the bitcount benchmark is shown in Figure 4c. This memory subsystem only provides a small performance improvement over the baseline (a speedup of less than 1%), but it also uses fewer block RAMs than the baseline memory subsystem (9 instead of 16). This feat is accomplished by splitting the address space between two caches. The first cache handles accesses to heap allocations whereas the second cache handles accesses to the stack. This type of split is common for the benchmarks that have accesses to a separate stack and heap.

Finally, we consider the memory subsystem for the jpegd benchmark, shown in Figure 4d. For the jpegd benchmark, the superoptimizer selected a split memory subsystem where only memory accesses to addresses 134513324 and higher go to a cache. This causes accesses to the program stack to be cached, but not accesses to heap allocations.

Of the superoptimized memory subsystems for the FPGA target, none contained only a single-level cache component. Five of the memory subsystems contained splits (bitcount, fft, jpegd, mm, and sha), five contained scratchpads (dijkstra, hash, heap, patricia, and qsort), and five contained address transformations (dijkstra, hash, mm, patricia, and qsort). Further, all of the superoptimized memory subsystems performed better than the baseline memory subsystem, even if only marginally better in some cases.

### 4.2 ASIC Results

The best-case speedup for the ASIC target is shown in Figure 6. For the ASIC target, we assume that, in the best case, all memory accesses hit a scratchpad with a 1-cycle access time and cycle time. Here we see that the geometric-mean best-case speedup is 17×. As in the FPGA case, it is not necessarily possible to achieve such a speedup.

The superoptimizer is able to get more impressive speedups for the ASIC than the FPGA for two reasons. First, the ASIC is assumed to be running at a higher clock frequency than the FPGA (1 GHz versus 250 MHz), making a miss in the memory subsystem have a greater impact. Second, there are more trade-offs for the ASIC memory components. In particular, when targeting an ASIC, the optimizer uses the access time and cycle time results from CACTI rather than using a fixed access time and cycle time as is done for the FPGA. Figure 7 shows the speedup that the optimized memory subsystem provides over the baseline memory subsystem. The geometric mean speedup is 6.52×.

The greatest increase in performance is again seen for the mm benchmark, whose memory subsystem is shown in Figure 5b. This memory subsystem has two sets of address rotations. The rotation by 27 bits causes every eighth entry of the first source matrix for 16384 entries to be stored in the first scratchpad, which has a cycle time of 1 cycle. Another 65536 entries of the first source matrix are stored in the second scratchpad, which has a cycle time of 3 cycles. Finally, the second set of rotations causes columns of the second source matrix to be cached in a way similar to the memory subsystem for the FPGA. Although the first address rotation may seem unnecessary, by reducing conflict misses in the cache, it actually improves the performance of the memory subsystem.

The memory subsystem for the hash benchmark targeting the ASIC is shown in Figure 5a. As is the case with the mm benchmark, the subsystem for the hash benchmark is similar to the subsystem for the FPGA. However, rather than an xor transform, this subsystem uses a rotate. In addition, this subsystem incorporates two scratchpads instead of one.

The memory subsystem discovered for the bitcount benchmark, shown in Figure 5c, is similar to the memory subsystem discovered for the bitcount for the FPGA target, shown in Figure 4c. Note that the split offset is only slightly different. However, here we have a cache before the split rather than on the left side of the split.

The last memory subsystem we consider in detail is the memory subsystem for the jpegd benchmark, shown in Figure 5d. This memory subsystem is one of the most complex memory subsystems discovered. The split causes access to the memory in the stack space to be mapped to a 4-level cache. Finally, accesses to both the stack and heap are backed by a smaller cache. The four levels of cache in the split each have slightly different properties and

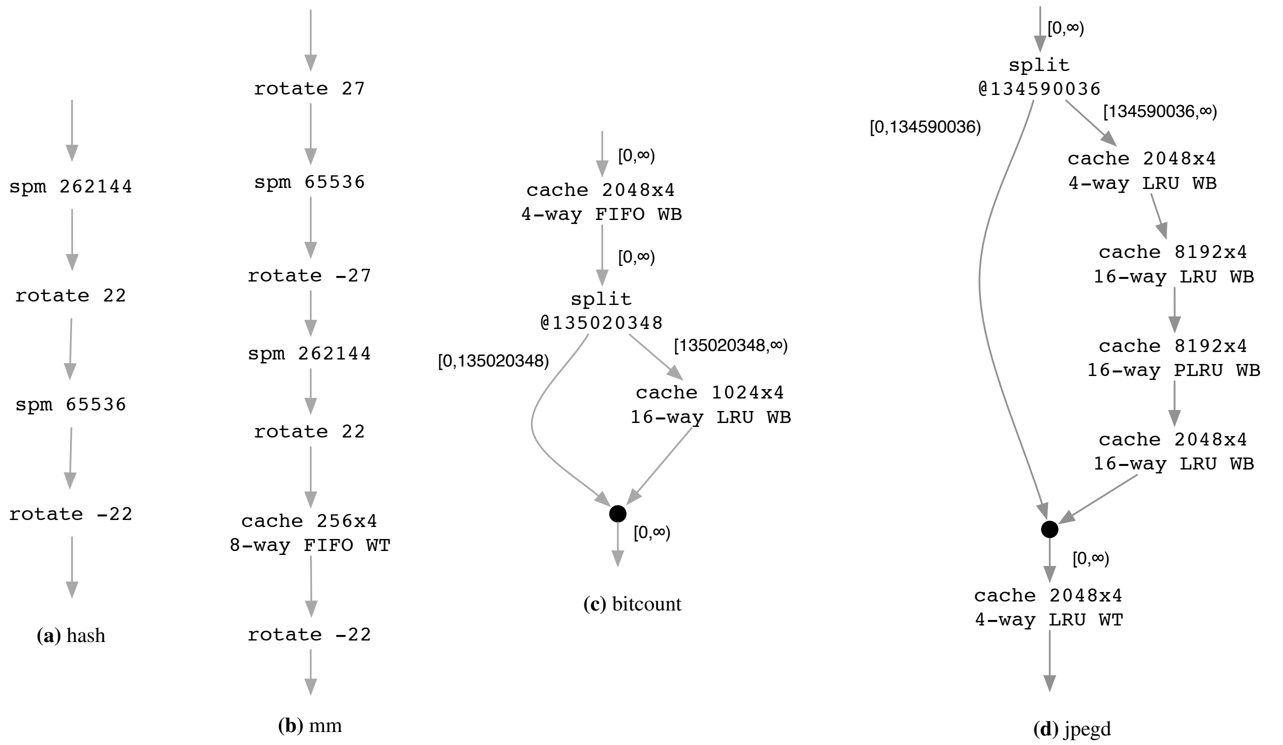


**Figure 5.** Superoptimized Memory Subsystems for the ASIC Target

removing any one of the caches causes a decrease in performance. Having separate, smaller caches such as this can be beneficial since smaller caches are faster than larger caches.

As is the case with the FPGA target, none of the superoptimized memory subsystems for the ASIC target contained only a single-level cache component. Four of the memory subsystems contained splits (bitcount, jpegd, patricia, and sha), six contained scratchpads (dijkstra, fft, hash, heap, mm, qsort) and six contained address transformations (dijkstra, fft, hash, heap, mm, and qsort). Further, like the FPGA target, all of the superoptimized memory subsystems performed better than the baseline memory subsystem.

## 4.3 Memory Subsystem Specificity

Finally, we consider how specific each of the memory subsystems is to the application for which the subsystem was superoptimized. Figure 8 shows a heat map comparing the results of running each of the 10 benchmarks with each of the 10 superoptimized memory subsystems for the FPGA target. The results are computed by dividing the total access time of each benchmark running with each memory subsystem by the total access time of the benchmark running with the memory subsystem that was superoptimized for that benchmark. In the figure, darker colors represent better performance.

In Figure 8, we see that the mm and heap benchmarks appear to run well only on the memory subsystems that are superoptimized for them. For the mm benchmark, the performance improvement from the rotate in the memory subsystem is significant enough to prevent any of the other memory subsystems from approaching the performance of the mm memory subsystem. The heap benchmark contains only a scratchpad, which causes accesses to the start of the heap, which are most frequent, to be fast. However, such a structure is suboptimal for the other benchmarks, though the hash benchmark performs fairly well with the memory subsystem for the heap benchmark. In all cases, the memory subsystem that was superoptimized for a particular benchmark provides the best performance for that benchmark.

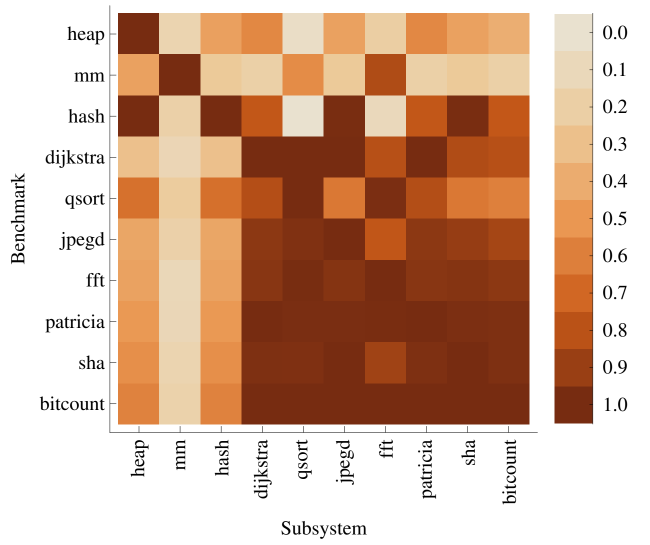


**Figure 8.** FPGA Memory Subsystem Specificity

Figure 9 shows a heat map comparing the results of running each of the 10 benchmarks with each of the 10 superoptimized memory subsystems for the ASIC target. As is the case with the FPGA results, the benchmarks all perform best with the superoptimized memory subsystem for the particular benchmark. In fact, the results are more specific for the ASIC target than for the FPGA tar-

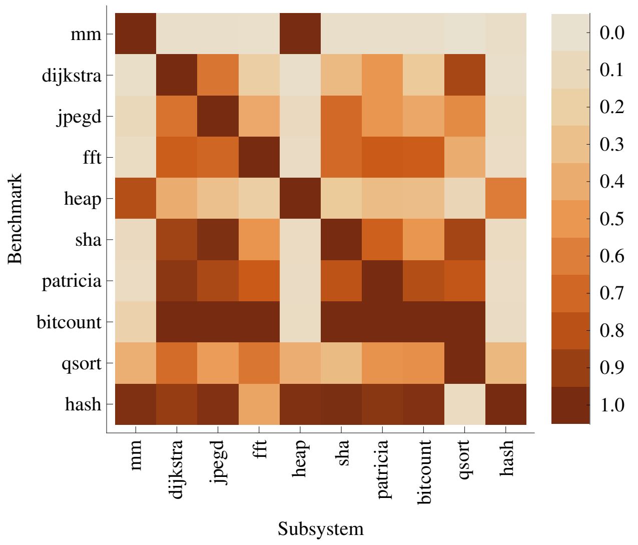


**Figure 9.** ASIC Memory Subsystem Specificity

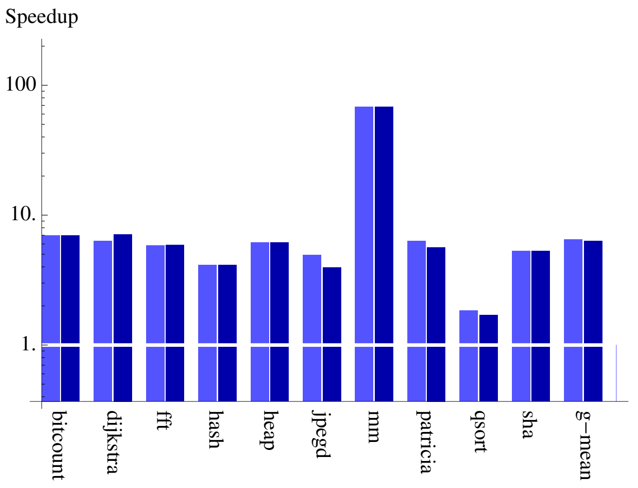


**Figure 10.** Speedup with Different Inputs

get, which is likely due to the fact that the ASIC target runs faster and has a more complex search space.

Given that the superoptimized memory subsystems are specific to the benchmark for which they were superoptimized, we note that the memory subsystem may further be specific to a particular run of the benchmark. To investigate this, we used a different input data set of the same size for each of the benchmarks for the ASIC target. For example, for the jpegd benchmark, a different input image of the same dimensions as the original was chosen. A comparison of the speedups over the baseline memory subsystem for the original data set and the new data set is shown in Figure 10.

In Figure 10, the lighter bars (on the left) show the speedup of the superoptimized memory subsystem over the baseline memory subsystem for the original data set and the darker bars (on the right) show the speedup for the modified data set. For many of the benchmarks there is little or no difference and in one case (dijkstra), the speedup actually improved. Overall, the geometric mean dropped from 6.43× to 6.27×. Although its impossible to draw anything conclusive from these results, it appears that the effects of over-fitting are minimal.

## 5. Discussion

When there are enough resources available on the target device (FPGA or ASIC) to contain the complete memory image of a benchmark, one might expect that a scratchpad memory would be the ideal choice. However, the addresses from these traces is do not necessarily start at 0 nor are they contiguous due to the memory layout of the application. Therefore, for such benchmarks, we might expect to see address transformations to move the bulk of the memory references into a scratchpad.

Unfortunately, the number of address transformations needed to move all memory accesses into a scratchpad can make it unlikely the optimizer will discover such a memory subsystem. In addition, the overhead, either in terms of area or BRAMs, of having many small scratchpads may prevent the optimizer from dividing up the memory addresses perfectly. Thus, a cache structure is often selected instead of a scratchpad. This is why, for example, the discovered memory subsystem for the bitcount benchmark uses caches for both the FPGA and ASIC targets despite the fact that the memory image of the bitcount benchmark could fit easily within a memory subsystem afforded by the constraints.

Despite the limitations of placing scratchpads, the optimizer uses such a technique for several of the benchmarks, most notably, the hash and heap benchmarks. By doing so, certain addresses that occur frequently can be accessed faster. Further, accesses that hit in a scratchpad that is before a cache avoid cache pollution.

An observation when comparing the superoptimized memory subsystems for the ASIC target against those for the FPGA target is that, for a particular benchmark, the overall structure is similar, though the memory subsystems for the ASIC target are typically more complex. We expect that the memory subsystems would have similar properties since the benchmark and main memory are held constant. Further, we expect that there would be some variation due to the different properties of the memory subsystem components when deployed on our target devices.

As a last point of discussion, we note that these memory subsystems are specific not only to an individual application, but also a particular run of that application. Although Section 4.3 demonstrates that this effect can be minimal, it is quite possible that some superoptimized memory subsystems could be overly-specific. Obviously, this can be bad in some cases. For example, when traversing a graph, we likely do not want the memory subsystem to have good performance only for a particular graph. On the other hand, when the memory accesses are not data dependent or when we expect the data to be similar from run to run (biased accesses into a hash table, for instance), it may be acceptable or even beneficial for a memory subsystem to be highly optimized for a particular access pattern.

## 6. Conclusion

We have shown that it is possible to superoptimize memory subsystems for specific applications that out-perform a general-purpose memory subsystem. Unlike previous work, the memory subsystems that our superoptimizer discovers can be arbitrarily complex and contain components other than simple caches. To superoptimize a memory subsystem, we use a form of threshold acceptance. We are then able to improve the discovery process by using information from the address trace.

This work targets both an FPGA as well as an ASIC process. For the FPGA target, we have validated the discovered memory subsystems by generating VHDL for each of the subsystems. The VHDL was then synthesized to ensure that the discovered memory subsystems are realizable at the required frequency. For the ASIC process, we used the CACTI [30] tool to get area and time estimates for each of the memory components.

In the future, we would like to extend this work to speed up the superoptimization process and support more memory subsystems. We would also like to investigate ways to find a representative trace signature for multiple runs of the same application to avoid discovering memory subsystems that are specific to a particular run. In addition, we would like to explore application-specific memory subsystems for parallel applications.

## 7. Acknowledgments

We thank our colleagues and the EIT staff for the use of the Engineering Cloud Cluster.

## References

[1] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. LEAP scratchpads: automatic memory and cache management for reconfigurable logic. In *Proc. of 19th ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, pages 25–28, 2011.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Trans. on Computers*, 52(10):1243–1258, Oct. 2003.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proc. of 8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.

[4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. of 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, 1994.

[5] J. Chang, P. Ranganathan, D. A. Roberts, M. A. Shah, and J. Sontag. Data storage apparatus and methods, Mar. 2012. US Patent App. 2012/0131278.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–24, 1999.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proc. of ACM Conf. on Programming Language Design and Implementation*, pages 1–12, 1999.

[8] G. Dueck and T. Scheuer. Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90(1):161–175, 1990.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of 40th Symp. on Foundations of Computer Science*, pages 285–297, 1999.

[10] A. Ghosh and T. Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Trans. on Design Automation of Electronic Systems*, 9(4):419–440, Oct. 2004.

[11] A. Gordon-Ross, F. Vahid, and N. Dutt. Automatic tuning of two-level caches to embedded applications. In *Proc. of the Conf. on Design, Automation and Test in Europe*, page 10208, 2004.

[12] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast configurable-cache tuning with a unified second-level cache. In *Proc. of Int'l Symp. on Low Power Electronics and Design*, pages 323–326, 2005.

[13] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 341–352, 1992.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of 4th Int'l Workshop on Workload Characterization*, pages 3–14, 2001.

[15] T. C. Hu, A. B. Kahng, and C.-W. A. Tsao. Old bachelor acceptance: A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7(4):417–425, 1995.

[16] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proc. of 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 195–206, 2006.

[17] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Int'l Symp. on Computer Architecture*, pages 364–373, 1990.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[19] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proc. of 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 185–194, 2006.

[20] H. Massalin. Superoptimizer: a look at the smallest program. In *Proc. of 2nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, 1987.

[21] A. Naz. *Split Array and Scalar Data Caches: A Comprehensive Study of Data Cache Organization*. PhD thesis, Univ. of North Texas, 2007.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 89–100, 2007.

[23] P. Panda, N. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):3–13, 1999.

[24] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. Performance and power of cache-based reconfigurable computing. *ACM SIGARCH Computer Architecture News*, 37(3):395–405, 2009.

[25] P. Ranjan Panda, N. D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. De Greef. Data memory organization and optimizations in application-specific systems. *IEEE Design & Test of Computers*, 18(3):56–68, 2001.

[26] Raspberry Pi. `http://www.raspberrypi.org`.

[27] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proc. of 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, 2013.

[28] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, Nov. 2002.

[29] K. T. Sundararajan, T. M. Jones, and N. P. Topham. Smart cache: A self adaptive cache architecture for energy efficiency. In *Proc. of Int'l Conf. on Embedded Computer Systems*, pages 41–50, 2011.

[30] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. *HP Laboratories*, 2, Apr. 2008.

[31] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proc. of 13th Int'l Conf. on Supercomputing*, pages 145–154, 1999.

[32] J. G. Wingbermuehle, R. D. Chamberlain, and R. K. Cytron. ScalaPipe: A streaming application generator. In *Proc. of 2012 Symp. on Application Accelerators in High-Performance Computing*, pages 244–254, July 2012.

[33] J. G. Wingbermuehle, R. K. Cytron, and R. D. Chamberlain. Optimization of application-specific memories. *Computer Architecture Letters*, Apr. 2013.

[34] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1): 20–24, Mar. 1995.

[35] C. Zhang and F. Vahid. Using a victim buffer in an application-specific memory hierarchy. In *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, pages 220–225, 2004.